Kapitel 5: Einführung in Pandas

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2. Auflage. Sebastopol, CA [u. a.]: O'Reilly.

Überarbeitet: armin.baenziger@zhaw.ch, 15. Januar 2020

- **Pandas** wird im weiteren Verlauf des Kurses die zentrale Bibliothek sein.
- Pandas enthält Datenstrukturen und Tools zur Datenbearbeitung, die in Python für eine schnelle und einfache Datenbereinigung und -analyse sorgen.
- Pandas verwendet wesentliche Teile von NumPys idiomatischem array-based Computing, insbesondere Array-basierte Funktionen und eine *Präferenz für die Datenverarbeitung ohne* `for` *-Schleifen*.
- Während Pandas viele Codierungs-Idiome von NumPy verwendet, ist der grösste Unterschied, dass Pandas für die Arbeit mit *tabellarischen* oder *heterogenen* Daten entwickelt wurde.
- Seit dem Open-Source-Projekt im Jahr 2010 ist Pandas zu einer grossen Bibliothek gereift, die in einer Vielzahl von Anwendungsfällen in der Praxis anwendbar ist.
- Pandas leitet sich überigens aus dem Wort "panel data" ab (https://de.wikipedia.org/wiki/Paneldaten (https://de.wikipedia.org/wiki/Paneldaten)).

In [1]:
```
%autosave 0
```

```
Autosave disabled
```

In [2]:
```
# Wichtige Bibliotheken mit üblichen Abkürzungen laden:
import numpy as np
import pandas as pd   # Usanz ist, pandas mit pd abzukürzen
```

Einführung Pandas-Datenstrukturen

Die zwei wichtigsten Datenstrukturen in Pandas sind **Series** und **DataFrames**.

Series

Eine *Series* setzt sich aus einem eindimensionalen *Werte-Array* und einem *Index* zusammen. Wenn man den Index nicht explizit bestimmt, ist dieser wie `range(n)` .

In [3]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[3]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

Man kann Werte-Array und Index mit den Attributen `values` und `index` separat erhalten.

In [4]:
```
obj.values        # Werte-NumPy-Array der Series obj
```

Out[4]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [5]:
```
obj.index        #  Index der Series obj (Sequenz)
```

Out[5]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [6]:
```
list(obj.index)  # Mit der Funktion list() erhält man die explizite Liste
```

Out[6]:
```
[0, 1, 2, 3]
```

Die Auswahl von Objekten in einer *Series* geschieht über den Index.

In [7]:
```
obj[0]
```

Out[7]:
```
4
```

In [8]:
```
obj[2]
```

Out[8]:
```
-5
```

In [9]:
```
obj[1:3]
# Slicing-Regel obj[Start:Stop]
# Ab Start bis zum letzten Wert vor Stop
```

Out[9]:
```
1    7
2   -5
dtype: int64
```

In [10]:
```
obj[:2]   # Erste zwei Werte.
```

Out[10]:
```
0    4
1    7
dtype: int64
```

In [11]:
```
obj[2:]   # Ab Posisiton 2 (dritter Wert) bis Ende.
```

Out[11]:
```
2   -5
3    3
dtype: int64
```

Kontrollfrage:

In [12]:
```
# Aufgabe: Slicen Sie die ersten drei Werte aus der Series `obj`.
obj[:3]
```

Out[12]:
```
0    4
1    7
2   -5
dtype: int64
```

Labels

Oft möchte man die Werte einer *Series* mit einem *Label* identifizieren.

In [13]:
```
obj2 = pd.Series([4, 7, -5], index=['Anna', 'Berta', 'Claudia'])
obj2
```

Out[13]:
```
Anna       4
Berta      7
Claudia   -5
dtype: int64
```

In [14]:
```
obj2['Berta']          # Wert, der zu "Berta" gehört.
```

Out[14]:
```
7
```

In [15]:
```
obj2['Berta'] = 6     # Wert zuweisen
obj2
```

Out[15]:
```
Anna       4
Berta      6
Claudia   -5
dtype: int64
```

In [16]:
```
# Mehrere Werte (in gewünschter Reihenfolge) ausgeben:
obj2[['Berta', 'Anna']]
```

Out[16]:
```
Berta    6
Anna     4
dtype: int64
```

In [17]:
```
# Welche Werte sind positiv?
obj2 > 0
```

Out[17]:
```
Anna        True
Berta       True
Claudia    False
dtype: bool
```

In [18]:
```
# Positive Werte auswählen:
obj2[obj2 > 0]
```

Out[18]:
```
Anna     4
Berta    6
dtype: int64
```

In [19]:
```
# Alle Werte verdoppeln:
obj2 * 2     # Wie bei NumPy. Index-Link bleibt bestehen.
```

Out[19]:
```
Anna        8
Berta      12
Claudia   -10
dtype: int64
```

In [20]:
```
# Man kann auch NumPy-Funktion auf Series anwenden:
np.exp(obj2)     # Exponentialfunktion aus NumPy-Bibliohek
```

Out[20]:
```
Anna        54.598150
Berta      403.428793
Claudia      0.006738
dtype: float64
```

In [21]:
```
'Anna' in obj2   # Prüfen, ob ein Index-Wert existiert.
```

Out[21]:
```
True
```

In [22]:
```
'Anne' in obj2
```

Out[22]:
```
False
```

Kontrollfragen:

In [23]:
```
# Gegeben:
obj2
```

Out[23]:
```
Anna       4
Berta      6
Claudia   -5
dtype: int64
```

In [24]:
```
# Frage 1: Was ist der Output?
obj2['Claudia']
```

Out[24]:
```
-5
```

In [25]:
```
# Frage 2: Was ist der Output?
np.abs(obj2['Claudia'])
```

Out[25]:
```
5
```

In [26]:
```
# Frage 3: Was ist der Output?
obj2[obj2!=6]
```

Out[26]:
```
Anna       4
Claudia   -5
dtype: int64
```

Ein *Dict* kann man sehr einfach in eine *Series* umwandeln.

In [27]:
```
flaeche_dict = {'GR': 7105, 'BE': 5959,
                'VS': 5225, 'VD': 3212}
flaeche_dict
```

Out[27]:
```
{'GR': 7105, 'BE': 5959, 'VS': 5225, 'VD': 3212}
```

In [28]:
```
flaeche = pd.Series(flaeche_dict)
flaeche    # Dict-Key wird zu Index
```

Out[28]:
```
GR    7105
BE    5959
VS    5225
VD    3212
dtype: int64
```

In [29]:
```
# Eigene Anordnung von Index mit entsprechenden Werten, falls
# diese im Dict vorhanden sind (Fehlwert, wenn der Index nicht
# im Dict ist):
flaeche2 = pd.Series(flaeche_dict,
                     index=['BE', 'GR', 'TI', 'VD', 'ZH'])
flaeche2
```

Out[29]:
```
BE    5959.0
GR    7105.0
TI       NaN
VD    3212.0
ZH       NaN
dtype: float64
```

- Falls vorhanden, werden die Werte den Indizes zugeordnet. Ansonsten gibt es Fehldaten (missing data), welche mit `NaN` gekennzeichnet sind.
- Die Funktionen/Methoden `isnull` und `notnull` werden in Pandas verwendet, um Fehldaten aufzuspüren.

In [30]:
```
flaeche2.isnull()
```

Out[30]:
```
BE    False
GR    False
TI     True
VD    False
ZH     True
dtype: bool
```

In [31]:
```
flaeche2.notnull()
```

Out[31]:
```
BE     True
GR     True
TI    False
VD     True
ZH    False
dtype: bool
```

In [32]:
```
# Wie viele Fehlwerte gibt es in flaeche2?
flaeche2.isnull().sum()
```

Out[32]:
```
2
```

Kontrollfrage:

In [33]:
```
# Frage: Was ist wohl der Output?
flaeche2[flaeche2.notnull()]
```

Out[33]:
```
BE    5959.0
GR    7105.0
VD    3212.0
dtype: float64
```

Fehldaten werden in Kapitel 7 ausführlicher behandelt.

Matching

Eine sehr nützliche Eigenschaft von *Series* ist, dass bei arithmetischen Operationen ein Matching bezüglich dem Index geschieht. Beispiel:

In [34]:
```
# Erste Series für Beispiel:
bevoelkerung2017 = {'BE': 1031126, 'GR': 197888, 'TI':  353709,
                    'VD':  793129, 'VS': 341463, 'ZH': 1504346}
bevoelkerung2017 = pd.Series(bevoelkerung2017)
bevoelkerung2017
```

Out[34]:
```
BE    1031126
GR     197888
TI     353709
VD     793129
VS     341463
ZH    1504346
dtype: int64
```

In [35]:
```
# Zweite Series für Beispiel:
bevoelkerung2018 = {'VD': 799145, 'BE': 1034977, 'VS': 343955,
                    'TI': 353343, 'ZH': 1520968, 'GR': 198379}
bevoelkerung2018 = pd.Series(bevoelkerung2018)
bevoelkerung2018
```

Out[35]:
```
VD     799145
BE    1034977
VS     343955
TI     353343
ZH    1520968
GR     198379
dtype: int64
```

Beachten Sie, dass die *Reihenfolge* der Kantone in den zwei Series unterschiedlich ist!

In [36]:
```
delta = bevoelkerung2018 - bevoelkerung2017
delta
# Werte mit gleichem Index werden verrechnet! Sehr gut!
```

Out[36]:
```
BE     3851
GR      491
TI     -366
VD     6016
VS     2492
ZH    16622
dtype: int64
```

In [37]:
```
# Relative Veränderung:
delta / bevoelkerung2017
```

Out[37]:
```
BE    0.003735
GR    0.002481
TI   -0.001035
VD    0.007585
VS    0.007298
ZH    0.011049
dtype: float64
```

Die ständige Wohnbevölkerung hat beispielsweise im Kanton Zürich um 1.1% zugenommen zwischen 2017 und 2018.

Der Series-Index kann verändert werden:

In [38]:
```
# Gegeben:
s = delta[:3].copy()
s.index = ['Bern', 'Graubünden', 'Tessin']
s
```

Out[38]:
```
Bern          3851
Graubünden     491
Tessin        -366
dtype: int64
```

Wir werden "sicherere" Verfahren kennenlernen, den Index zu verändern!

DataFrame

- Ein *DataFrame* ist eine Datentabelle, welche eine Sammlung von Spalten (Variablen) aufweist, bei der im Gegensatz zu einem Numpy-Array jede Spalte einen *unterschiedlichen Datentypen* aufweisen kann. Innerhalb einer Spalte ist der Datentyp aber homogen.
- Das DataFrame hat sowohl einen *Zeilen-* als auch einen *Spaltenindex*.
- Es gibt viele Möglichkeiten, ein DataFrame zu erstellen. Häufig verwendet man einen Dict von *gleichlangen* Listen oder NumPy-Arrays. Im Lehrmittel finden Sie eine komplette Liste mit möglichen Inputs für den DataFrame-Konstruktor.
- *Üblicherweise werden DataFrames aber aus importieren Daten erzeugt, wie wir noch sehen werden.*

In [39]:
```
daten = {'Stadt': ['Biel', 'Biel', 'Biel',
                   'Thun', 'Thun', 'Thun'],
         'Jahr': [2016, 2017, 2018]*2,
         'ALQ': [5.1, 5.0, 3.9, 2.6, 2.6, 2.0]}
daten   # ein Dict
```

Out[39]:
```
{'Stadt': ['Biel', 'Biel', 'Biel', 'Thun', 'Thun', 'Thun'],
 'Jahr': [2016, 2017, 2018, 2016, 2017, 2018],
 'ALQ': [5.1, 5.0, 3.9, 2.6, 2.6, 2.0]}
```

In [40]:
```
df = pd.DataFrame(daten)  # df ist unser erstes DataFrame
df
```

Out[40]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

Ein DataFrame kann mit einer Excel-Tabelle verglichen werden.

In [41]:
```
df.head()    # Anzeige der ersten 5 Zeilen (Default).
# Hier nicht sehr nützlich, da der Datensatz nur 6 Zeilen enthält.
```

Out[41]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6

In [42]:
```
df.head(3)  # Anzeige der ersten 3 Zeilen.
```

Out[42]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9

In [43]:
```
df[:3]      # So ginge es auch.
```

Out[43]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9

In [44]:
```
df.tail()   # Anzeige der letzten 5 Zeilen (Default).
```

Out[44]:

	Stadt	Jahr	ALQ
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

In [45]:
```
df[-5:]      # So ginge es auch.
```

Out[45]:

	Stadt	Jahr	ALQ
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

Wichtig: Wie wählen wir Spalten (Variablen) aus?

In [46]:
```
df['Stadt']   # Ausgabe einer Spalte mit der "Dict-Notation"
```

Out[46]:
```
0    Biel
1    Biel
2    Biel
3    Thun
4    Thun
5    Thun
Name: Stadt, dtype: object
```

In [47]:
```
df.Stadt # Ausgabe einer Spalte durch Attribut (Tab-Ergänzung möglich!)
# Manchmal praktisch, aber mit eingeschränkter Funktionalität (siehe
# weiter unten).
```

Out[47]:
```
0    Biel
1    Biel
2    Biel
3    Thun
4    Thun
5    Thun
Name: Stadt, dtype: object
```

Kontrollfragen:

In [48]:
```
# Gegeben:
df
```

Out[48]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

In [49]:
```
# Aufgabe 1: Wählen Sie die ALQ aus df aus.
df['ALQ']    # oder df.ALQ
```

Out[49]:
```
0    5.1
1    5.0
2    3.9
3    2.6
4    2.6
5    2.0
Name: ALQ, dtype: float64
```

In [50]:
```
# Aufgabe 2: Was ist der Output?
df.tail(1)
```

Out[50]:

	Stadt	Jahr	ALQ
5	Thun	2018	2.0

Das Attribut `columns` liefert die Spaltenüberschriften im DataFrame:

In [51]:
```
df.columns
```

Out[51]:
```
Index(['Stadt', 'Jahr', 'ALQ'], dtype='object')
```

Die Spaltenüberschriften können nachträglich verändert werden.

In [52]:
```
# Spaltennamen (Variablennamen) ändern: Möglichkeit 1
df.columns = ['Stadt', 'Jahr', 'Arbeitslosenquote']
df
```

Out[52]:

	Stadt	Jahr	Arbeitslosenquote
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

In [53]:
```
# Spaltennamen (Variablennamen) ändern: Möglichkeit 2
df.rename(columns = {'Arbeitslosenquote': 'ALQ'}, inplace=True)
# mit inplace=True sind die Änderungen permament
df
```

Out[53]:

	Stadt	Jahr	ALQ
0	Biel	2016	5.1
1	Biel	2017	5.0
2	Biel	2018	3.9
3	Thun	2016	2.6
4	Thun	2017	2.6
5	Thun	2018	2.0

Die zweite Möglichkeit ist der ersten vorzuziehen. Warum?

In [54]:
```
# Zeilenindex ändern:
df.index = df.Jahr
df
```

Out[54]:

	Stadt	Jahr	ALQ
Jahr			
2016	Biel	2016	5.1
2017	Biel	2017	5.0
2018	Biel	2018	3.9
2016	Thun	2016	2.6
2017	Thun	2017	2.6
2018	Thun	2018	2.0

Die Variable `Jahr` ist nun redundant und kann gelöscht werden. (Wir werden später eine elegantere Möglichkeit kennenlernen, einen Index zu setzen.)

In [55]:
```
del df['Jahr']     # Löschen der Spalte Jahr.
# del df.Jahr ginge übrigens nicht!
```

- Das Jahr ist nun keine Spalte (Variable) mehr und wird mit `df.index` angesprochen.
- Wir werden später eine bessere Möglichkeit kennenlernen, Spalten (und Zeilen) zu löschen.

Zeilen können mit `loc` über den Index*label* und mit `iloc` über die Index*position* angesprochen werden. Beispiele:

In [56]:
```
df.loc[2016]           # Auswahl über ZeilenLABEL
```

Out[56]:

	Stadt	ALQ
Jahr		
2016	Biel	5.1
2016	Thun	2.6

In [57]:
```
# Mehrere Labels können als Liste übergeben werden:
df.loc[[2016, 2018]]
```

Out[57]:

Jahr	Stadt	ALQ
2016	Biel	5.1
2016	Thun	2.6
2018	Biel	3.9
2018	Thun	2.0

In [58]:
```
df.iloc[0]            # Auswahl über ZeilenPOSITION
```

Out[58]:
```
Stadt    Biel
ALQ       5.1
Name: 2016, dtype: object
```

In [59]:
```
df.iloc[[0, 3]]
```

Out[59]:

Jahr	Stadt	ALQ
2016	Biel	5.1
2016	Thun	2.6

Kontrollfragen:

In [60]:
```
# Gegeben:
df
```

Out[60]:

Jahr	Stadt	ALQ
2016	Biel	5.1
2017	Biel	5.0
2018	Biel	3.9
2016	Thun	2.6
2017	Thun	2.6
2018	Thun	2.0

In [61]:
```
# Frage 1: Was ist der Output?
df.loc[2018]
```

Out[61]:

Jahr	Stadt	ALQ
2018	Biel	3.9
2018	Thun	2.0

In [62]:
```
# Frage 2: Was ist der Output?
df.iloc[[2, 5]]
```

Out[62]:

Jahr	Stadt	ALQ
2018	Biel	3.9
2018	Thun	2.0

In [63]:
```
# Frage 3: Was ist wohl der Output?
df.loc[2018, 'ALQ']
```

Out[63]:
```
Jahr
2018    3.9
2018    2.0
Name: ALQ, dtype: float64
```

Spalten/Variblen hinzufügen:

In [64]:
```
# Ausgangslage:
df
```

Out[64]:

Jahr	Stadt	ALQ
2016	Biel	5.1
2017	Biel	5.0
2018	Biel	3.9
2016	Thun	2.6
2017	Thun	2.6
2018	Thun	2.0

In [65]:
```
df['Bevoelkerung'] = [54456, 54640, None, 43568, 43743, None]
# Länge muss konform sein (korrekte Länge haben)
df
```

Out[65]:

Jahr	Stadt	ALQ	Bevoelkerung
2016	Biel	5.1	54456.0
2017	Biel	5.0	54640.0
2018	Biel	3.9	NaN
2016	Thun	2.6	43568.0
2017	Thun	2.6	43743.0
2018	Thun	2.0	NaN

In [66]:
```
# Generierung einer Variablen aus einer anderen:
df['ALQhoch'] = df.ALQ >= 5
df
```

Out[66]:

Jahr	Stadt	ALQ	Bevoelkerung	ALQhoch
2016	Biel	5.1	54456.0	True
2017	Biel	5.0	54640.0	True
2018	Biel	3.9	NaN	False
2016	Thun	2.6	43568.0	False
2017	Thun	2.6	43743.0	False
2018	Thun	2.0	NaN	False

Beachten Sie, dass für die Erstellung einer neuen Spalte/Variable auf der rechten Seite die Dict-Notation stehen muss (`df.ALQhoch = df.ALQ >= 5` hätte nicht funktioniert).

Kontrollfragen:

In [67]:
```
# Aufgabe: Generieren Sie die Variable ALQtief, welche True
# ist, falls die ALQ höchstens 3 Prozent ist.
df['ALQtief'] = df.ALQ <= 3
df
```

Out[67]:

Jahr	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

Exkurs: Es ist auch möglich, dem Index und den Spalten eine Überschrift zu geben:

In [68]:
```
df.index.name = 'Jahr'         # existiert hier so schon
df.columns.name = 'Variablen'
df
```

Out[68]:

Jahr	Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

Achsen (Zeilen oder Spalten) löschen: `drop`-Methode

Series

In [69]:
```
serie = flaeche.copy()    # Kopie erstellen
serie
```

Out[69]:
```
GR    7105
BE    5959
VS    5225
VD    3212
dtype: int64
```

In [70]:
```
serie.drop('BE')
```

Out[70]:
```
GR    7105
VS    5225
VD    3212
dtype: int64
```

In [71]:
```
serie       # BE wurde nicht permanent gelöscht!
```

Out[71]:
```
GR    7105
BE    5959
VS    5225
VD    3212
dtype: int64
```

In [72]:
```
serie.drop('BE', inplace=True) # Jetzt ist BE permanent gelöscht!
serie
```

Out[72]:
```
GR    7105
VS    5225
VD    3212
dtype: int64
```

In [73]:
```
serie.drop(['GR', 'VD'])      # mehrere Einträge löschen
```

Out[73]:
```
VS    5225
dtype: int64
```

DataFrame

In [74]:
```
# Ausgangslage
df
```

Out[74]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

In [75]:
```
df.drop([2017]) # Zeilen löschen (mit inplace=True permanent)
```

Out[75]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2018	Thun	2.0	NaN	False	True

In [76]:
```
df.drop('Bevoelkerung', axis=1)
```

Out[76]:

Variablen	Stadt	ALQ	ALQhoch	ALQtief
Jahr				
2016	Biel	5.1	True	False
2017	Biel	5.0	True	False
2018	Biel	3.9	False	False
2016	Thun	2.6	False	True
2017	Thun	2.6	False	True
2018	Thun	2.0	False	True

Mit `axis=0` (Default) sind Zeilen gemeint und mit `axis=1` Spalten. Alternativ kann man statt `axis=1` auch `axis='columns'` verwenden.

In [77]:
```
df.drop(['ALQtief'], axis='columns') # Alternative zu axis=1
```

Out[77]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch
Jahr				
2016	Biel	5.1	54456.0	True
2017	Biel	5.0	54640.0	True
2018	Biel	3.9	NaN	False
2016	Thun	2.6	43568.0	False
2017	Thun	2.6	43743.0	False
2018	Thun	2.0	NaN	False

Alternativ kann man eine Spalte mit `del` (permanent) löschen. Wie wir weiter oben gesehen haben.

Kontrollfragen:

In [78]:
```
# Aufgabe 1: Fügen Sie df die Spalte "i" mit den Werten
# 1 bis 6 hinzu.
df['i'] = range(1,7)
df
```

Out[78]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief	i
Jahr						
2016	Biel	5.1	54456.0	True	False	1
2017	Biel	5.0	54640.0	True	False	2
2018	Biel	3.9	NaN	False	False	3
2016	Thun	2.6	43568.0	False	True	4
2017	Thun	2.6	43743.0	False	True	5
2018	Thun	2.0	NaN	False	True	6

In [79]:
```
# Aufgabe 2: Löschen Sie die Variable "i" wieder mit der
# `drop`-Methode permanent.
df.drop('i', axis=1, inplace=True)
df
```

Out[79]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

(Nochmals) Indexierung, Selektion und Filtering

Series

In [80]:
```
flaeche        # gegeben
```

Out[80]:
```
GR    7105
BE    5959
VS    5225
VD    3212
dtype: int64
```

In [81]:
```
flaeche[1:3]  # Slicen
```

Out[81]:
```
BE    5959
VS    5225
dtype: int64
```

Slicing mit Labels funktioniert anders als das übliche Python-Slicing: Endpunkte sind inklusiv.

In [82]:
```
flaeche['BE':'VS']
```

Out[82]:
```
BE    5959
VS    5225
dtype: int64
```

DataFrames

In [83]:
```
df  # gegeben
```

Out[83]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

In [84]:
```
df[:2]    # Erste zwei Zeilen wählen.
```

Out[84]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False

In [85]:
```
df[df.Stadt=='Thun']  # Nur Daten von Thun selektieren
```

Out[85]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

Beachten Sie, dass in der eckigen Klammer nochmals angegeben werden muss, aus welchem Dataframe die Variable "Stadt" stammt.

Exkurs: Alternativ könnte man die Methode `query` verwenden.

In [86]:
```
df.query("Stadt=='Thun'")
```

Out[86]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

Selektion mit loc und iloc

Für die Selektion von Zeilen und Spalten aus DataFrames stehen in Pandas die zwei Methoden `loc` (Achsenlabels) und `iloc` (Position in Achse) zur Verfügung.

In [87]:
```
# Ausgangslage:
df
```

Out[87]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

In [88]:
```
df.loc[2016]    # wie zuvor gesehen
```

Out[88]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2016	Thun	2.6	43568.0	False	True

In [89]:
```
# Auswahl von Zeilen und Spalten über Label:
df.loc[2016, 'ALQ']
```

Out[89]:
```
Jahr
2016    5.1
2016    2.6
Name: ALQ, dtype: float64
```

In [90]:
```
df.loc[2016, ['Stadt', 'ALQ']]
```

Out[90]:

Variablen	Stadt	ALQ
Jahr		
2016	Biel	5.1
2016	Thun	2.6

In [91]:
```
# Lösung über Positionen in den Indizes:
df.iloc[[0,3], [0,1]]
```

Out[91]:

Variablen	Stadt	ALQ
Jahr		
2016	Biel	5.1
2016	Thun	2.6

In [92]:
```
df.iloc[2]      # Ergebnis ist eine Series
```

Out[92]:
```
Variablen
Stadt            Biel
ALQ               3.9
Bevoelkerung      NaN
ALQhoch         False
ALQtief         False
Name: 2018, dtype: object
```

In [93]:
```
df.iloc[[2]]   # Ergebnis ist ein DataFrame
```

Out[93]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2018	Biel	3.9	NaN	False	False

`loc` und `iloc` funktionieren auch mit Slices:

In [94]:
```
df  # zur Erinnerung
```

Out[94]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

In [95]:
```
df.iloc[3:, :2]
# df ab Zeilenposition 3 und Spalten bis unter Position 2.
```

Out[95]:

Variablen	Stadt	ALQ
Jahr		
2016	Thun	2.6
2017	Thun	2.6
2018	Thun	2.0

Kontrollfragen:

In [96]:
```
# Gegeben:
df
```

Out[96]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2018	Thun	2.0	NaN	False	True

In [97]:
```
# Frage 1: Was ist der Output?
df.loc[2016, 'Stadt':'ALQ']
```

Out[97]:

Variablen	Stadt	ALQ
Jahr		
2016	Biel	5.1
2016	Thun	2.6

In [98]:
```
# Frage 2: Was ist der Output?
df.iloc[3:, :2]
```

Out[98]:

Variablen	Stadt	ALQ
Jahr		
2016	Thun	2.6
2017	Thun	2.6
2018	Thun	2.0

Funktionen auf Spalten, Zeilen oder alle Elemente anwenden

NumPy ufuncs (element-wise array methods) funktionieren auch mit Pandas-Objekten.

In [99]:
```
# Beispieldaten:
df2 = pd.DataFrame({'X': [-4, 3, 0],
                    'Y': [2, -1, 5]},
                   index=list('abc'))
df2
```

Out[99]:

	X	Y
a	-4	2
b	3	-1
c	0	5

In [100]:
```
df2.max(axis=0) # Maximum pro Spalte (entlang Zeilen)
```

Out[100]:
```
X    3
Y    5
dtype: int64
```

In [101]:
```
df2.max()       # axis=0 ist der Default, es geht also ohne!
```

Out[101]:
```
X    3
Y    5
dtype: int64
```

In [102]:
```
# Exkurs:
df2.max(axis=1) # Maximum pro Zeile (entlang Spalten)
```

Out[102]:
```
a    2
b    3
c    5
dtype: int64
```

In [103]:
```
df2.mean()      # Arithmetisches Mittel pro Spalte
```

Out[103]:
```
X   -0.333333
Y    2.000000
dtype: float64
```

In [104]:
```
df2.std()      # Standardabweichung pro Spalte
```

Out[104]:
```
X    3.511885
Y    3.000000
dtype: float64
```

Es ist auch möglich, NumPy-Funktionen auf Spalten anzuwenden:

In [105]:
```
np.abs(df2)    # Absolutwerte/Beträge
```

Out[105]:

	X	Y
a	4	2
b	3	1
c	0	5

Kontrollfragen:

In [106]:
```
# Gegeben:
df2
```

Out[106]:

	X	Y
a	-4	2
b	3	-1
c	0	5

In [107]:
```
# Frage 1: Was ist der Output?
df2.median()
```

Out[107]:
```
X    0.0
Y    2.0
dtype: float64
```

In [108]:
```
# Frage 2: Was ist der Output?
df2['Y'].sum()
```

Out[108]:
```
6
```

Mit der Methode `apply` kann man eine (eigene) *Funktion* auf jede Spalte (oder Zeile) anwenden.

In [109]:
```
# Definition einer eigenen Funktion:
def spannweite(x):
    return x.max() - x.min()
```

In [110]:
```
# Mit apply eigene Funktion auf DataFrame-Spalten anwenden:
df2.apply(spannweite)
```

Out[110]:
```
X    7
Y    6
dtype: int64
```

Kontrollfragen:

In [111]:
```
# Aufgabe 1: Schreiben Sie die Funktion schiefemass(werte),
# welche die Differenz  mean(werte) - median(werte)  zurückgibt.
def schiefemass(werte):
    return werte.mean() - werte.median()
```

In [112]:
```
# Aufgabe 2: Wenden Sie die eben definierte Funktion schiefemass()
# auf die Spalten von df2 an.
df2.apply(schiefemass)
```

Out[112]:
```
X   -0.333333
Y    0.000000
dtype: float64
```

Sortieren

Indizes werden mit der Methode `sort_index` sortiert.

In [113]:
```
flaeche   # Beispiel einer Series
```

Out[113]:
```
GR    7105
BE    5959
VS    5225
VD    3212
dtype: int64
```

In [114]:
```
flaeche.sort_index()    # permanent mit inplace=True
```

Out[114]:
```
BE    5959
GR    7105
VD    3212
VS    5225
dtype: int64
```

In [115]:
```
# Beispiel eines DataFrame:
df.sort_index()   # Zeilenindex sortieren (Default)
```

Out[115]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Biel	5.1	54456.0	True	False
2016	Thun	2.6	43568.0	False	True
2017	Biel	5.0	54640.0	True	False
2017	Thun	2.6	43743.0	False	True
2018	Biel	3.9	NaN	False	False
2018	Thun	2.0	NaN	False	True

In [116]:
```
df.sort_index(axis=1)  # Spaltenüberschriften sortieren
```

Out[116]:

Variablen	ALQ	ALQhoch	ALQtief	Bevoelkerung	Stadt
Jahr					
2016	5.1	True	False	54456.0	Biel
2017	5.0	True	False	54640.0	Biel
2018	3.9	False	False	NaN	Biel
2016	2.6	False	True	43568.0	Thun
2017	2.6	False	True	43743.0	Thun
2018	2.0	False	True	NaN	Thun

In [117]:
```
df.sort_index(ascending=False)  # Index absteigend sortieren
```

Out[117]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2018	Biel	3.9	NaN	False	False
2018	Thun	2.0	NaN	False	True
2017	Biel	5.0	54640.0	True	False
2017	Thun	2.6	43743.0	False	True
2016	Biel	5.1	54456.0	True	False
2016	Thun	2.6	43568.0	False	True

Werte werden mit **`sort_values`** sortiert.

Bei Series:

In [118]:
```
flaeche.sort_values()
```

Out[118]:
```
VD    3212
VS    5225
BE    5959
GR    7105
dtype: int64
```

Bei DataFrames:

In [119]:
```
# DataFrame nach Werten der Spalte "Bevoelkerung" sortieren:
df.sort_values(by='Bevoelkerung')
# Fehlwerte (NaN) werden ans Ende sortiert.
```

Out[119]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True
2016	Biel	5.1	54456.0	True	False
2017	Biel	5.0	54640.0	True	False
2018	Biel	3.9	NaN	False	False
2018	Thun	2.0	NaN	False	True

In [120]:
```
# Zuerst nach Spalte "Stadt", dann "ALQ" sortieren
df.sort_values(by=['Stadt', 'ALQ'])
```

Out[120]:

Variablen	Stadt	ALQ	Bevoelkerung	ALQhoch	ALQtief
Jahr					
2018	Biel	3.9	NaN	False	False
2017	Biel	5.0	54640.0	True	False
2016	Biel	5.1	54456.0	True	False
2018	Thun	2.0	NaN	False	True
2016	Thun	2.6	43568.0	False	True
2017	Thun	2.6	43743.0	False	True

Kontrollfrage:

In [121]:
```
# Gegeben:
df2
```

Out[121]:

	X	Y
a	-4	2
b	3	-1
c	0	5

In [122]:
```
# Aufgabe: Sortieren Sie das DataFrame df2 nach der Variable Y.
df2.sort_values(by='Y')
```

Out[122]:

	X	Y
b	3	-1
a	-4	2
c	0	5

Daten zusammenfassen / deskriptive Statistiken

Pandas-Objekte sind mit vielen **mathematischen und statistischen Methoden** versehen. Die meisten davon sind sogenannte "Reductions" bzw. zusammenfassende Statistiken, welche aus einer Series oder Spalte/Zeile eines DataFrame einen einzigen Wert extrahieren. Wir haben bereits einige davon oben gesehen, z. B. `sum()` oder `max()` .

In [123]:
```
# Ausgangslage: Beispieldaten erstellen
np.random.seed(777)
data = np.random.randint(-2, 3, size=(4,3))
df3 = pd.DataFrame(data, columns=list('ABC'))
df3['D'] = df3.A >= 0
df3['E'] = list('cbba')
df3
```

Out[123]:

	A	B	C	D	E
0	1	-1	2	True	c
1	-1	0	-2	False	b
2	0	-2	1	True	b
3	-1	0	2	False	a

In [124]:
```
df3.sum()
```

Out[124]:
```
A       -1
B       -3
C        3
D        2
E     cbba
dtype: object
```

Hinweise:

- Die Summe in Spalte `D` entspricht der Anzahl `True`.
- Die Summe in Spalte `E` ist hier nicht sinnvoll!

Eine Stärke von Pandas ist der Umgang mit Fehlwerten (`NaN`). Um dies zu demonstrieren, setzen wir zwei Fehlwerte in den Datensatz.

In [125]:
```
# Zwei Fehlwerte (NaN) erzeugen:
df3.iloc[1,2] = None    # entweder None
df3.iloc[2,3] = np.nan  # oder mit np.nan
df3
```

Out[125]:

	A	B	C	D	E
0	1	-1	2.0	1.0	c
1	-1	0	NaN	0.0	b
2	0	-2	1.0	NaN	b
3	-1	0	2.0	0.0	a

- Beachten Sie, dass durch das Einsetzen eines Fehlwertes in Spalte `D` `True` zu 1 und `False` zu 0 wurde.
- `NaN` werden bei Berechnungen "übersprungen" (skipped), ausser alle Werte in der Spalte (oder Zeile) sind `NaN`. Mit der Option `skipna = False` kann man diesen Default übersteuern. Beispiele:

In [126]:
```
df3.mean()      # Spaltenmittelwerte, NaN nicht berücksichtigen
```

Out[126]:
```
A   -0.250000
B   -0.750000
C    1.666667
D    0.333333
dtype: float64
```

In [127]:
```
df3.mean(skipna=False)  # NaN, falls mind. 1 NaN in der Spalte
```

Out[127]:
```
A   -0.25
B   -0.75
C     NaN
D     NaN
dtype: float64
```

Des Weiteren gibt es Methoden, welche akkumulieren.

In [128]:
```
df3.cumsum()    # Die Spaltenwerte werden aufkummuliert.
```

Out[128]:

	A	B	C	D	E
0	1	-1	2	1	c
1	0	-1	NaN	1	cb
2	0	-3	3	NaN	cbb
3	-1	-3	5	1	cbba

Schliesslich gibt es Methoden, die mehrere Statistiken liefern. Eine wichtige ist `describe` .

In [129]:
```
# Output der Methode describe bei metrischen Daten:
df3.describe()
```

Out[129]:

	A	B	C	D
count	4.000000	4.000000	3.000000	3.000000
mean	-0.250000	-0.750000	1.666667	0.333333
std	0.957427	0.957427	0.577350	0.577350
min	-1.000000	-2.000000	1.000000	0.000000
25%	-1.000000	-1.250000	1.500000	0.000000
50%	-0.500000	-0.500000	2.000000	0.000000
75%	0.250000	0.000000	2.000000	0.500000
max	1.000000	0.000000	2.000000	1.000000

Eläuterungen:

`count` : Anzahl Werte pro Spalte (ohne NaN)
`mean` : Arithmetische Mittelwerte pro Spalte
`std` : Standardabweichungen pro Spalte
`min` : Minimum pro Spalte
`25%` : Erstes Quartil
`50%` : Zweites Quartil = Median
`75%` : Drittes Quartil
`max` : Maximum pro Spalte

In [130]:
```
# Output der Methode describe bei nicht metrischen Daten:
df3['E'].describe()
```

Out[130]:
```
count     4
unique    3
top       b
freq      2
Name: E, dtype: object
```

Eläuterungen:

`count` : Anzahl Werte (ohne NaN)
`unique` : Anzahl unterschiedlicher Werte
`top` : Modus
`freq` : Häufigkeit des Modus

Kovarianz und Korrelation

Kovarianz und Korrelation werden aus Daten*paaren* berechnet.

In [131]:
```
# Beispieldaten mit Aktienkursen aus dem Verzeichnis examples laden
# (Genauere Erklärungen dazu folgen im nächsten Kapitel):
price = pd.read_pickle('../examples/yahoo_price.pkl')
price.head()
```

Out[131]:

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	27.990226	313.062468	113.304536	25.884104
2010-01-05	28.038618	311.683844	111.935822	25.892466
2010-01-06	27.592626	303.826685	111.208683	25.733566
2010-01-07	27.541619	296.753749	110.823732	25.465944
2010-01-08	27.724725	300.709808	111.935822	25.641571

Falls die Daten nicht geladen werden, stimmt der Pfad oben nicht.
`../examples/yahoo_price.pkl` bedeutet, dass Python zuerst ein Verzeichnis höher geht (..) und dann in das Verzeichnis `examples` wechselt, in dem die Datei `yahoo_price.pkl` liegen sollte. Details folgen später ...

Methode `pct_change` :
Mit der Methode (Abkürzung für *percentage change*) können relative Veränderungen pro Spalte berechnet werden.

In [132]:
```
returns = price.pct_change()
returns.head()
# In der ersten Zeile stehen NaN, da die Renditen zum Vortag nicht
# berechnet werden können.
```

Out[132]:

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	NaN	NaN	NaN	NaN
2010-01-05	0.001729	-0.004404	-0.012080	0.000323
2010-01-06	-0.015906	-0.025209	-0.006496	-0.006137
2010-01-07	-0.001849	-0.023280	-0.003462	-0.010400
2010-01-08	0.006648	0.013331	0.010035	0.006897

In [133]:
```
# Kovarianz der Renditen zwischen Microsoft (MSFT) und IBM:
returns['MSFT'].cov(returns['IBM'])
```

Out[133]: 8.870655479703546e-05

In [134]:
```
# Korrelation der Renditen zwischen Microsoft (MSFT) und IBM:
returns['MSFT'].corr(returns['IBM'])
```

Out[134]: 0.4997636114415114

In [135]:
```
returns.MSFT.corr(returns.IBM)    # alternative Schreibweise
```

Out[135]: 0.4997636114415114

Mit den DataFrame-Methoden `cov` und `corr` erhält man die ganze Kovarianz- bzw. Korrelationsmatrix.

In [136]:
```
returns.corr()
```

Out[136]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

Erläuterung: Die Korrelation der Renditen zwischen Google (`GOOG`) und Microsoft (`MSFT`) beträgt 0.465919. Die anderen Werte sind gleich zu lesen.

In [137]:
```
# Nur Korrelationen aller Aktien mit GOOG berechnen:
returns.corrwith(returns.GOOG)
```

Out[137]:
```
AAPL    0.407919
GOOG    1.000000
IBM     0.405099
MSFT    0.465919
dtype: float64
```

Die Korrelation von Google (GOOG) mit sich selber ist natürlich 1.

Kontrollfragen:

In [138]:
```
# Gegeben:
returns.corr()
```

Out[138]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

In [139]:
```
# Frage 1: Was ist der Output?
returns.AAPL.corr(returns.IBM)
```

Out[139]:
```
0.38681743611391
```

In [140]:
```
# Gegeben:
Kurse = pd.DataFrame({'KursA': [100, 120, 150],
                      'KursB': [100, 90, 100]},
                     index = [2016, 2017, 2018])
Kurse
```

Out[140]:

	KursA	KursB
2016	100	100
2017	120	90
2018	150	100

In [141]:
```
# Frage: Was ist der Output?
Kurse.pct_change()
```

Out[141]:

	KursA	KursB
2016	NaN	NaN
2017	0.20	-0.100000
2018	0.25	0.111111

Unikate, Häufigkeiten und Zugehörigkeiten

In [142]:
```
obj = pd.Series(list('abbcbac'))
obj
```

Out[142]:
```
0    a
1    b
2    b
3    c
4    b
5    a
6    c
dtype: object
```

In [143]:
```
obj.unique()           # unterschiedliche Werte
```

Out[143]:
```
array(['a', 'b', 'c'], dtype=object)
```

In [144]:
```
len(obj.unique())     # Anzahl unterschiedlicher Werte
```

Out[144]:
```
3
```

In [145]:
```
obj.nunique()         # einfacher mit Pandas nunique()
```

Out[145]:
```
3
```

In [146]:
```
obj.value_counts()     # Häufigkeitsverteilung: WICHTIGE Funktion!
```

Out[146]:
```
b    3
c    2
a    2
dtype: int64
```

In [147]:
```
obj.value_counts(normalize=True)    # relative Häufigkeitsverteilung
```

Out[147]:
```
b    0.428571
c    0.285714
a    0.285714
dtype: float64
```

Per Default sortiert Pandas nach absteigender Häufigkeit. Manchmal möchte man aber nach dem Index sortieren. Zwei mögliche Lösungen dafür:

In [148]:
```
# Möglichkeit 1: Argument sort=False setzen
obj.value_counts(sort=False)
```

Out[148]:
```
a    2
b    3
c    2
dtype: int64
```

In [149]:
```
# Möglichkeit 2: Nachträglich den Index sortieren
obj.value_counts().sort_index()
```

Out[149]:
```
a    2
b    3
c    2
dtype: int64
```

Kontrollfragen:

In [150]:
```
# Gegeben:
np.random.seed(543)
# 100 Würfe mit fairem Würfel simulieren:
augen = pd.Series(np.random.randint(1,7,100))
augen.head()  # die ersten 5 Realisationen
```

Out[150]:
```
0    2
1    2
2    5
3    2
4    6
dtype: int32
```

In [151]:
```
# Aufgabe 1: Erstellen Sie die absolute Häufigkeitsverteilung von "augen".
augen.value_counts()
```

Out[151]:
```
1    22
2    18
5    17
3    16
4    15
6    12
dtype: int64
```

In [152]:
```
# Aufgabe 2: Erstellen Sie die relative Häufigkeitsverteilung von "augen".
# Sortieren Sie nach dem Index (nicht nach der Häufigkeit).
augen.value_counts(normalize=True, sort=False)
```

Out[152]:
```
1    0.22
2    0.18
3    0.16
4    0.15
5    0.17
6    0.12
dtype: float64
```

Fazit

- In diesem Kapitel haben wir einige Grundlagen von Pandas erarbeitet, die wir im Verlauf des Kurses immer wieder nutzen werden.
- Im nächsten Kapitel diskutieren wir Tools zum Lesen und Schreiben von Daten in Pandas.